Implementing Dense Optical Flow Computation on a Heterogeneous FPGA SoC in C

WENJIE CHEN, East China Normal University

ZHIBIN WANG, QIN WU, JIUZHEN LIANG, and ZHILEI CHAI, Jiangnan University

High-quality optical flow computation algorithms are computationally intensive. The low computational speed of such algorithms causes difficulties for real-world applications. In this article, we propose an optimized implementation of the classical Combine-Brightness-Gradient (CBG) model on the Xilinx ZYNQ FPGA-SoC, by taking advantage of the inherent algorithmic parallelism and ZYNQ architecture. The execution time decreases to 0.82 second with a lower power consumption (1.881W). It is better than software implementation on PC (Intel i7-3520M, 2.9GHz), which costs 2.635 seconds and 35W. We use C rather than HDLs to describe the algorithm for rapid prototyping.

CCS Concepts: • **Hardware** → **Hardware accelerators**; **Hardware-software codesign**; • **Computing methodologies** → **Image representations**;

Additional Key Words and Phrases: Variational optical flow computation, FPGA-accelerated heterogeneous SoC, ZYNQ, high-level synthesis (HLS)

ACM Reference Format:

Wenjie Chen, Zhibin Wang, Qin Wu, Jiuzhen Liang, and Zhilei Chai. 2016. Implementing dense optical flow computation on a heterogeneous FPGA SoC in C. *ACM Trans. Archit. Code Optim.* 13, 3, Article 25 (August 2016), 25 pages.

DOI: <http://dx.doi.org/10.1145/2948976>

1. INTRODUCTION

Optical flow is the pattern of apparent motion of objects, surfaces, and edges in a visual scene that is caused by the relative motion between an observer and the scene. Optical flow computation estimates motion by extracting either instantaneous image velocities or discrete image displacements. Performance evaluations [Baker et al. 2011; Barron et al. 1994; Horn and Schunck 1981] have shown that variational methods are capable of computing dense optical flow fields. Moreover, variational methods can preserve motion boundaries [Black and Anandan 1996] and perform favorably in the presence of noise and occlusions [Bruhn et al. 2005a; Sundberg et al. 2011]. Recent improvements can even handle large displacements gracefully [Brox et al. 2009; Brox and Malik 2011].

This work is supported by the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing, NSF of China under grants 60703106, 61170121, and 61202312; the 111 Project under grant B12018; the National High Technology Research and Development Program of China (a.k.a “863 program”) under grant 2015AA015304; and the Shanghai Natural Science Foundation under grant 15ZR1410000.

Authors’ addresses: W. Chen, MoE Engineering Research Center for Software/Hardware Co-design Technology and Application, East China Normal University, Shanghai, China, 200062; email: wjchen@sei.ecnu.edu.cn; Z. Wang, Q. Wu, J. Liang, and Z. Chai, School of Internet of Things Engineering, Jiangnan University, Wuxi, China 214122; emails: stepzhibin@jiangnan.edu.cn, qinwu@jiangnan.edu.cn, jzliang@jiangnan.edu.cn, zlchai@jiangnan.edu.cn.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2016 ACM 1544-3566/2016/08-ART25 \$15.00

DOI: <http://dx.doi.org/10.1145/2948976>

However, high-quality variational algorithms for dense optical flow computation are computationally intensive, which results in long execution times and high-energy consumption. For example, on a general PC with an Intel i7-3520M (2.9GHz) central processing unit (CPU), the popular linear variational method Combine-Brightness-Gradient (CBG) [Brox et al. 2004; Bruhn et al. 2005b] takes 2.635 seconds to compute the optical flow between two 640×480 frames at a power of 35W, which means that 92.3J of energy is consumed. In general, an embedded processor uses less power than a PC CPU. In our experiment, which was performed on an ARM Cortex-A9 processor within a ZYNQ System-on-Chip (SoC) [ZYNQ 2015], the power consumption is 1.308W. However, computing the optical flow of the previous condition takes 26.68 seconds. This slow speed prevents optical flow computations in real-world applications, particularly in embedded systems such as vision-aided robots [Chao et al. 2014; Honegger et al. 2012], unmanned aerial vehicles [Herissé et al. 2012; Lippiello and Siciliano 2012] and so forth.

A considerable amount of recent research has focused on high-speed computation of optical flow. Some efforts focus on software optimization by improving the algorithms, schemes, and approaches, whereas the other efforts focus on hardware acceleration. There are essentially two approaches for hardware acceleration: one is through a GPU, and the other is through a field-programmable gate array (FPGA). The related works are thoroughly introduced in Section 2. The main disadvantage of graphics processing unit (GPU) acceleration is its high power consumption, which is not suitable for embedded applications.

In contrast to CPUs and GPUs, FPGAs are able to support different types of fine-grained parallelism and provide highly efficient computation for optical flow algorithms. Nevertheless, implementing algorithms on pure FPGAs is time-consuming due to difficulties in debugging and driver issues when low-level programming languages such as VHDL/Verilog are used. Therefore, although more accurate algorithms such as CBG have already been developed, to the best of our knowledge, all the previous FPGA acceleration implementations are still based on the classical Horn and Schunck [1981] (HS) algorithm because CBG is more complicated than HS. This complexity arises because the former exploits both the first-order and second-order derivatives, whereas the latter only uses the first-order derivatives.

Fortunately, heterogeneous SoCs with different types of computing cores have emerged in recent years [Brodtkorb et al. 2010], such as the Xilinx ZYNQ [ZYNQ 2015] and Altera SoC FPGA [Altera 2015]. As introduced in Chung et al. [2010], the heterogeneous SoC has the potential to achieve better energy efficiency by combining traditional processors with unconventional cores. Furthermore, the CPUs in heterogeneous SoCs relieve users from the FPGA driver issue. In addition, high-level synthesis techniques such as AutoESL [Berkeley Design Technology, Inc. 1994] integrated into Xilinx development tools raise the level of abstraction by using C/C++ code in the system's description.

In this article, we establish a general workflow for the variational optical flow computation for both HS and CBG, as well as for other variational optical flow models. We implement this workflow on a Xilinx ZYNQ FPGA SoC. By exploiting the inherent parallelism of the algorithms and the architecture of the FPGA SoC (Z-7020CLG484), the CBG execution time decreased to 0.82 second, which achieved an acceleration of $32 \times$ over the pure software implementation (26.68 seconds) running on a processor on ZYNQ (ARM Cortex A9, 667MHz) for 640×480 images. We use C rather than Hardware Description Languages (HDL) to implement the optical flow algorithm on the FPGA SoC, which relieves engineers of nontrivial low-level details. Moreover, based on the same FPGA platform, we compare different models and study the relationship among models, architectures, computation accuracy, and overall performance.

The remainder of this article is structured as follows. Section 2 introduces the related works on variational optical flow algorithms and their acceleration. Section 3 describes the common workflow of variational optical flow computation, including the state-of-the-art optical flow algorithms. Section 4 analyzes different types of parallelism in optical flow computation and introduces our techniques to improve computational performance one by one. Section 5 introduces the key points in the development of both hardware and software. Section 6 presents and analyzes the overall experimental results of our FPGA-accelerated optical flow system and compares it with other accelerators in terms of various aspects. Section 7 draws the conclusions.

2. BACKGROUND AND RELATED WORKS

To obtain insights regarding acceleration in variational optical flow computation, in this section, we first briefly provide an overview of the variational optical flow and classify the different algorithms into three models for further analysis and comparison. Then, we discuss different works on software optimization and hardware acceleration, including GPU and FPGA. Finally, we introduce the architecture of ZYNQ, which is a heterogeneous SoC that is used as the hardware platform for optimization in this paper.

2.1. Advances in Variational Optical Flow Algorithms and Their Categories

Optical flow is a quantitative metric of the visual movement between consecutive image frames, which can be obtained by establishing correspondence between pixels in two frames. To do that, Horn and Schunck (HS) [Horn and Schunck 1981] proposed two widely used assumptions to compute the pixel correspondence. **Assumption 1** (*Brightness constant constraint*): the brightness of a pixel does not change between frames. **Assumption 2** (*Global smoothness constraint*): the optical flow vector varies smoothly over the image frames. With these assumptions, variational optical flow algorithms can be formulated as an energy minimization problem, in which the energy function comprises of a data term and a smoothness term. To better describe the optical flow smoothness, Lucas and Kanade [1981] proposed another assumption, which is as follows. **Assumption 3** (*Local constant optical flow constraint*): the optical flow vectors within a small neighborhood are more-or-less constant. Assumptions 1 and 3 are used by Lucas and Kanade (LK) to establish the energy function. Bruhn et al. combined the three assumptions proposed by HS and LK and proposed the *Combined-Local-Global* (CLG) model for a better robustness of optical flow computation under noisy condition [Bruhn and Weickert 2005]. Brox et al. introduced another assumption as a complement for the brightness constraint. **Assumption 4** (*Constant gradient constraint*): the gradient of the image brightness remains constant over image frames despite the pixel displacement. By combining assumptions 1, 2, and 4, Brox et al. [2004] proposed the *Combined-Brightness-Gradient* (CBG) model. Since constant gradient constraint is quite susceptible to brightness changes, which often appear in natural scenes, CBG is more suitable for the situations of natural scenes that slight changes in brightness occurs, particularly for scenarios where the camera is moved by different frames.

Moreover, introducing robustness into the energy function are deeply discussed. For instance, Black and Anandan [1996] presented a framework based on robust estimation that addresses violations of the brightness constancy and spatial smoothness assumptions, such as transparency, specular reflections, shadows, or fragmented occlusions. Some works used the isotropic smoothness term to preserve motion discontinuities [Brox et al. 2009; Bruhn et al. 2005a; Zimmer et al. 2011; Xu et al. 2012]. Bruhn et al. [2005a] imposed separate penalties on the brightness and gradient constraints. Zimmer et al. [2011] and Xu et al. [2012] further employed the normalized brightness and gradient constraints in optical flow computation. Recently, techniques to address

large displacements in optical flow estimation have also been introduced [Brox et al. 2009; Brox and Malik 2011].

Although many different techniques have emerged in the field of variational optical flow computation, from a general perspective, these techniques can be categorized into three models: *the HS model*, *the CLG model*, and *the CBG model*. The majority of the state-of-the-art techniques are enhancements based on these models, such as the nonquadratic robust data term (NQ-D) [Bruhn et al. 2005a], the nonquadratic robust smoothness term (NQ-S) [Xu et al. 2012; Zimmer et al. 2011], the multiscale strategy [Black and Anandan 1996; Bruhn and Weickert 2005], the warping strategy for handling large displacements [Brox et al. 2004, 2009; Brox and Malik 2011], and the segmentation strategy [Amiaz and Kiryati 2006]. Furthermore, for all of the variants belonging to different models, there are only two schemes for solving the energy functions: either the linear solver or the nonlinear solver [Bruhn et al. 2005b].

In this article, we discuss the general workflows for these three models and propose how to implement and optimize them on a heterogeneous FPGA SoC.

2.2. Accelerating Computation by Software and Hardware

As introduced in Section 1, accelerating optical flow computation is critical for real-world applications, whether by software optimization or by hardware acceleration (with GPU or FPGA).

2.2.1. Software Optimization. From the aspect of algorithms, Black and Anandan [1996] employed the hierarchical approach to reduce the computational complexity, which is considered in many modern applications. Bruhn and Weickert [2005] employed unidirectional and bidirectional multigrid schemes in hierarchical approaches, which outperform the classical scheme even with small problem sizes. Additionally, Gwosdek et al. [2010] introduced a multigrid red-black relaxation scheme for a parallel implementation of the linear CLG method.

2.2.2. Hardware Acceleration with GPU. Mizukami and Tadamura [2007] realized the HS algorithm on a GeForce GTX 8800. Using this approach requires 0.443 seconds to calculate the optical flow of an image with a resolution of 316×252 . Gwosdek et al. [2010] implemented a better optical flow on a GeForce GTX 285, which costs 0.98 seconds per frame for an image with a resolution of 640×480 . Sundaram et al. [2010] realized an optical flow method that can handle large displacements on a GeForce GTX 480 [Sundaram et al. 2010], and the processing time is 1.84 seconds. More GPU implementations of optical flow methods are introduced in Zimmer et al. [2011] and Xu et al. [2012]. The speed-up ratio of GPU-based acceleration is very high. However, the power consumption is also very high. For example, the GeForce GTX 285 consumes 205W in idle mode and 316W in full-speed mode. The high power consumption limits its applications, particularly in embedded systems.

2.2.3. Hardware Acceleration with FPGA. As introduced in Section 1, FPGAs that support different types of fine-grained parallelisms should receive more attention, particularly for embedded systems with power constraints. Arribas and Macia [2003] realized the HS algorithm on the Altera EP20K100; for images of 50×50 , the speed is 19 frames per second (fps) for 3 iterations. Martin et al. [2005] implemented the HS algorithm on the Altera APEX20K; the speed is 60fps for 256×256 images, but the number of iterations was not mentioned. Rustam et al. [2012] proposed a hardware architecture by combining the integer and fraction arithmetic functions. However, he also did not report the number of iterations. Gultekin and Saranli [2013] accelerated the HS algorithm on the Altera Cyclone, which achieved 257fps for images of 256×256 , and the power consumption was as low as 0.844W.

Fig. 1. Overview of the ZYNQ architecture.

Bahar et al. implemented the HS algorithm on FPGA (Altera Cyclone II), as reported in Bahar and Karimian [2012]. In their implementation, parallel storages are used to accelerate the computation of partial derivative, motion vector, and averaging motion vectors, while each of these computations on 1 pixel costs only 1 clock. Moreover, the implementation computes a very sparse optical flow to achieve a high frames per second: every 10th pixel is computed for a 240×320 pixels frame, which corresponds to an effective resolution of 23×31 pixels. Kunz et al. [2014] proposed an architecture to split iteration into four pipelined phases in order to reduce the calculations needed for future iteration stages. On an Altera Stratix, this implementation is able to obtain 640×512 optical flow at 30fps. Komorkiewicz et al. [2014] presented an efficient hardware implementation of the HS algorithm on the state-of-the-art FPGA [Komorkiewicz et al. 2014]. In their implementation, all iterations are extracted and pipelined in the FPGA in order to accelerate the computation. It can process 60fps of a $1,920 \times 1,080$ image on Xilinx Virtex 7. The disadvantage is that it requires considerable FPGA resources and limits the number of iterations.

The advantages of FPGA are the flexible architecture, high speed and lower power consumption. Nevertheless, implementing algorithms on pure FPGAs is time-consuming due to the difficulties in debugging and low-level programming languages such as VHDL/Verilog generally being used. Therefore, to the best of our knowledge, only basic dense optical flow algorithms (HS model) with low accuracy or sparse optical flow algorithms have been implemented on FPGA-based platforms.

2.3. Overview of ZYNQ Architecture

As introduced in Chung et al. [2010], the heterogeneous SoC has the potential to achieve better energy efficiency by combining traditional processors with unconventional cores. Xilinx released the first FPGA-accelerated heterogeneous SoC: ZYNQ [ZYNQ 2015]. Figure 1 illustrates the architecture of ZYNQ, which integrates an ARM-based *Processing System* (PS) and a *Programmable Logic* (PL) into a single chip. Four *AXI High-Performance ports* (HP) are used by the PL to access memory and to transfer data to and from the PS. Thus, computation-intensive algorithms or modules with good parallelism can be accelerated by being restructured and migrated to PL. The results computed by the PL are then sent back to the PS through AXI-VDMA (*Video Direct Memory Access*). In the PL, an AXI-VDMA core is required to handle the image data based on the AXI-Stream protocol, which connects accelerators in the PL and the HPs in the PS. This is the *data path*. The control of VDMA cores is programmed through the AXI4-Lite interface connected to *General-Purpose I/O ports* (GPIO) in the PS, which is the *control path*.

Fig. 2. General workflow of variational optical flow computation.

3. WORKFLOW OF VARIATIONAL OPTICAL FLOW COMPUTATION

In this section, we discuss the general workflow of variational optical flow computation; describe the framework of variational optical flow computation, which consists of three classical models (HS, CLG, and CBG); and introduce the relationship among these models and the relationship between the linear and nonlinear solvers.

As shown in Figure 2, variational optical flow algorithms generally consist of the following processing stages.

- **Presmoothing (S1):** This stage reduces the influence of noise and outliers by smoothing the input image sequences, generally by convolving images with smoothing filters.
- **Derivative computing (S2):** This stage computes spatial-temporal derivatives. Higher-order derivatives are considered in some methods to make the flow results more accurate. For example, as shown in Figure 2, the first-order derivative is used in the HS and CLG models, whereas both the first- and second-order derivatives are used in the CBG model. Computing the first-order derivatives obtains the spatial derivatives (E_x , E_y) and temporal derivative (E_t) from the two smoothed images. For the second-order derivatives, only the spatial derivatives of E_x , E_y , and E_t are computed, and five second-order derivative results (E_{xx} , E_{xy} , E_{yy} , E_{tx} , and E_{ty}) are obtained.
- **Motion tensor construction (S3):** This stage generates five motion tensors (J_{11} , J_{12} , J_{13} , J_{22} , and J_{23}) regardless of which optical flow model is adopted. Table I presents the methods for constructing the motion tensors for HS, CLG, and CBG. For HS, only the first-order derivatives (E_x , E_y , and E_t) are used directly. For CLG,

Table I. Motion Construction Model of HS, CLG, and CBG

HS	CLG	CBG
$J_{11} = E_x E_x$	$J_{11} = K_\sigma * E_x E_x$	$J_{11} = \gamma_1 E_x E_x + \gamma_2 (E_{xx} E_{xx} + E_{xy} E_{xy})$
$J_{12} = E_x E_y$	$J_{12} = K_\sigma * E_x E_y$	$J_{12} = \gamma_1 E_x E_y + \gamma_2 (E_{xy} E_{xx} + E_{yy} E_{xy})$
$J_{13} = E_x E_t$	$J_{13} = K_\sigma * E_x E_t$	$J_{13} = \gamma_1 E_x E_t + \gamma_2 (E_{xy} E_{tx} + E_{xy} E_{ty})$
$J_{22} = E_y E_y$	$J_{22} = K_\sigma * E_y E_y$	$J_{22} = \gamma_1 E_y E_y + \gamma_2 (E_{xy} E_{xy} + E_{yy} E_{yy})$
$J_{23} = E_y E_t$	$J_{23} = K_\sigma * E_y E_t$	$J_{23} = \gamma_1 E_y E_t + \gamma_2 (E_{xy} E_{tx} + E_{yy} E_{ty})$

an additional 2D least-squares convolution operation is required, where K_σ is the kernel of the least-squares operation [Bruhn et al. 2005b]. For CBG, the first-order derivatives and second-order derivatives (E_{xx} , E_{xy} , E_{yy} , E_{tx} , and E_{ty}) are combined to compute the five final motion tensors, where γ_1 and γ_2 denote the data weights.

- **Iteration scheme (S4):** This stage computes flow results through iterations. For the linear iteration scheme, only flow tensors (u, v) need to be updated. The iteration body employing the linear Successive Over Relaxation (SOR) is given by Equation (1), where i denotes the number of pixels in rows; j denotes the number of pixels in columns; J denotes the constructed motion tensors; u and v denote horizontal and vertical flow tensors, respectively; N_l denotes the two-neighborhood of the point (i, j) along l ; N_l^- and N_l^+ denote the right bottom and the top left neighbors, respectively; α denotes the smoothness weight parameter; and k denotes the iteration number.

$$\begin{aligned}
 [u]_{i,j}^{k+1} &= (1 - w)[u]_{i,j}^k + w \frac{\alpha \left(\sum_{(\tilde{i}, \tilde{j}) \in N_l^-(i,j)} \frac{[u]_{\tilde{i},\tilde{j}}^{k+1}}{h_l^2} + \sum_{(\tilde{i}, \tilde{j}) \in N_l^+(i,j)} \frac{[u]_{\tilde{i},\tilde{j}}^k}{h_l^2} \right) - ([J_{12}]_{i,j} [v]_{i,j}^k + [J_{13}]_{i,j})}{[J_{11}]_{i,j} + \alpha \sum_{(\tilde{i}, \tilde{j}) \in N_l(i,j)} \frac{1}{h_l^2}} \\
 [v]_{i,j}^{k+1} &= (1 - w)[v]_{i,j}^k + w \frac{\alpha \left(\sum_{(\tilde{i}, \tilde{j}) \in N_l^-(i,j)} \frac{[v]_{\tilde{i},\tilde{j}}^{k+1}}{h_l^2} + \sum_{(\tilde{i}, \tilde{j}) \in N_l^+(i,j)} \frac{[v]_{\tilde{i},\tilde{j}}^k}{h_l^2} \right) - ([J_{12}]_{i,j} [u]_{i,j}^{k+1} + [J_{23}]_{i,j})}{[J_{22}]_{i,j} + \alpha \sum_{(\tilde{i}, \tilde{j}) \in N_l(i,j)} \frac{1}{h_l^2}} \quad (1)
 \end{aligned}$$

For the nonlinear scheme, both flow tensors (u, v) and motion tensors (J_{11} , J_{12} , J_{13} , J_{22} , J_{23}) need to be updated. Although the data values of iteration schemes for motion and flow tensors may be different, the data interface for motion tensors and flow tensors is the same (i.e., five motion tensors and two flow tensors). Hence, the nonlinear iteration scheme can be extended from the linear iteration scheme by computing ψ and updating the motion tensors accordingly.

4. OPTIMIZING OPTICAL FLOW COMPUTATION BASED ON ZYNQ

Although HLS tools such as Xilinx Vivado are able to optimize conventional C code for FPGA platforms to some extent, we still have to understand the algorithm and the platform architecture for further optimization. This section takes CBG as an example to analyze the techniques for optimization. First, we analyze the parallelisms that exist in optical flow computation. Then, we propose different approaches step by step to restructure and optimize CBG based on the algorithm's intrinsic parallelisms and hardware architecture.

4.1. Parallelisms in Variational Optical Flow Methods

The parallelisms in CBG include task parallelism, data parallelism, and pipeline parallelism, which provide the basis for restructuring C code for FPGAs.

Fig. 3. Task parallelisms within CBG.

4.1.1. Task Parallelism. Figure 3 shows the workflow of the entire CBG model. As shown in Figure 3, data dependency exists in the operations of adjacent stages. The result of one stage is used as the input for the next stage. Thus, operations belonging to different stages have to compute one after the other. Fortunately, multiple operations belonging to the same stage can be computed independently. Assume that two frames of images are needed each time to compute the optical flow between them. As shown in Figure 3, presmoothing of two input images ($img1$ and $img2$) can be processed simultaneously to obtain two smoothed images ($simg1$ and $simg2$, respectively). Furthermore, the first derivatives (E_x , E_y) are computed based on the average of the two smoothed images $((simg1 + simg2)/2)$. E_t can be computed by $simg1 - simg2$. All of the derivatives can be computed independently and simultaneously. Similarly, the second-order derivatives E_{xx} , E_{xy} , E_{yy} , E_{tx} , and E_{ty} can be computed simultaneously. Finally, the motion tensors J_{11} , J_{12} , J_{13} , J_{22} , and J_{23} can also be computed simultaneously according to the right part of Table I. Thus, it is straightforward to accelerate operations within each stage by computing them as parallel tasks. The parallelism of the iteration body will be thoroughly discussed in Section 4.2.

4.1.2. Data Parallelism. In general, the operations of presmoothing and derivative computing are computed by convolution, which contains a typical type of data parallelism. Furthermore, for each motion tensor construction, multiple pairs of data can be computed by the same instruction. It is also a type of data parallelism that can be efficiently implemented on massively parallel hardware.

4.1.3. Pipeline Parallelism. Although data dependency exists in different stages, as mentioned earlier, and operations in different stages have to be computed one after the other, it is not necessary to wait for all results from the previous stages to be available to start the next stages. For each stage, only the pixels of the beginning rows and columns should be available to start the operation. Thus, when suitable hardware is available, all stages of CBG can work simultaneously in a pipeline fashion. However, in stage S4, the SOR iteration requires a number of iterations to generate the final result, which destroys the pipeline of the entire system. According to our experiment, as shown in Figure 12, the stage of SOR iteration accounts for the largest portion of the overall execution time. Hence, this stage should be optimized first to smooth the pipeline of the entire system. This will be discussed in detail in Section 4.2.

4.2. Optimization Techniques

In this section, techniques to optimize optical flow computation based on ZYNQ are introduced. First, computation-intensive C modules are restructured and migrated

Fig. 4. Data dependency of SOR and Red-Black SOR scheme. (a) SOR; (b) Red-Black SOR.

to FPGAs for acceleration. Then, techniques to improve the overall throughput by optimizing cooperation between the PS and the PL of ZYNQ are discussed.

4.2.1. Optimizing SOR Iteration on PL. As mentioned in Section 4.1.3, the stage of SOR iteration is the bottleneck of the entire system. Three techniques are used to optimize this stage: (1) accelerating the iteration body of SOR on PL (i.e., FPGAs) with a **parallel hardware structure**; (2) decreasing the number of iterations and memory accessing via **loop unrolling**; and (3) further decreasing the requirement for memory bandwidth by **extending the iteration body** to contain more pipeline stages.

(1) *Implementing parallelized iteration body on PL.* The basic concept is to accelerate the iteration body of SOR on PL. However, according to the original SOR described in Equation (1), for each u or v under processing, its four neighbors are involved. For example, as shown in Figure 4(a), assuming that element 7 is under processing, elements 2, 6, 8, and 12 are needed for the computation. Note that according to the scheme of SOR, the left and top neighbors (i.e., 2 and 6) have been updated, whereas the right and bottom neighbors (i.e., 8 and 12) have not been updated. Because these two neighbors have to be updated before the current element is processed, it results in data dependency that prevents elements from being processed in parallel.

To avoid the data dependency shown in Figure 4(a), an improved derivation called *Red-Black SOR* was proposed by Evans [1984]. As shown in Figure 4(b), this derivation uses a two-pass scheme to replace the one-pass scheme in the original SOR. All the elements are divided into red elements and black elements, and all the red elements are adjacent to black neighbors and vice versa. In the first pass, the red elements are processed while all the black elements are kept untouched. Then, in the second pass, all the black elements are processed while their neighbors (i.e., the red elements) are kept untouched because they were updated in the previous pass. This scheme makes the elements under processing and their neighbors update in different passes. Thus, theoretically, during each pass, all elements can be processed in parallel because their neighbors are always available.

Therefore, the Red-Black SOR with abundant parallelism is suitable to be implemented on PL for acceleration. Figure 5 shows the hardware structure of the Red-Black SOR. It is implemented by HLS, which is restructured from the conventional C code of optical flow computation. The initial flow tensor inu , inv and the motion tensors J_{11} , J_{12} , J_{13} , J_{22} , and J_{23} are retrieved from memory. The intermediate results of optical flow $outu$, $outv$ after each iteration are written back to the memory. Then, $outu$, $outv$ are used as inu , inv for the next iteration.

To make four neighbors of the current u or v available simultaneously, three buffers and a 3×3 window are used, as shown in Figure 5. For example, $Buf_u.0$, $Buf_u.1$ and $Buf_u.2$ are connected together in sequence. Initially, inu is brought into $Buf_u.0$, then spilled over to $Buf_u.1$ and $Buf_u.2$ successively. Once these three buffers are filled, the

Fig. 5. Hardware structure of the Red-Black Scheme.

three elements at the end of each buffer are simultaneously pushed into the window. Thus, after some cycles of delay, the window is able to provide four neighbors for each u simultaneously. As shown in Equation (1), only the basic operations of addition, subtraction, multiplication, and division are used for computing u and v , provided that inu , inv , and J are available. These operations can be implemented in hardware in a straightforward manner. Because u is used to compute v during the same iteration, u should be available before v , and several buffers are needed for the synchronization of u and v . By using the structure shown in Figure 5, each group of u , v , J_{11} , J_{12} , J_{13} , J_{22} , and J_{23} can be processed within constant cycles (e.g., five) after an initial delay during a single iteration. Finally, a pair of flow tensors $outu$, $outv$ can be obtained within constant cycles after an initial delay.

(2) *Loop unrolling of SOR iteration.* In this section, loop unrolling is used to further accelerate the computation. Loop unrolling is an efficient approach for improving instruction-level parallelism for the pipelined processor. For CBG executed on ZYNQ, there are two advantages to unrolling the loop of S4: (1) multiple iteration bodies can be executed simultaneously in a pipelined fashion through obtaining data from the former directly, as shown in Figure 6(a); and (2) more importantly, by decreasing the number of iterations controlled by PS, it can reduce memory access. The data interface between PL and memory contains five motion tensors and two flow tensors, a total of 224 bits (7×32), which results in time-consuming memory access. Therefore, fewer iterations means less total execution time.

(3) *Extending the iteration body.* As shown in Figure 2, five motion tensors ($J_{11} \dots J_{23}$) are generated after S3. If only S4 is taken as the iteration body executed on PL, then these tensors should be saved in memory in advance. For each iteration, five motion tensors and two flow tensors, 224 bits in total, need to be fetched from memory, which takes a large portion of the total execution time. To reduce the memory bandwidth requirement, S1 to S3 are also taken into the iteration body, as shown in Figure 6(b). For each iteration, the original images $img1$ and $img2$ and the flow tensors u and v , a total of 128 bits, need to be fetched from memory. The values of the derivatives and

Fig. 6. Optimization of pipeline. (a) Optimization by loop unrolling; (b) Optimization by extending iteration body.

motion tensors are all computed on the fly during each iteration. Because multiple stages are pipelined, the time to compute motion tensors on the fly is considerably shorter than that of memory access caused by a larger bandwidth.

4.2.2. Optimizing Other Stages of CBG. In addition to the S4 stage, S1, S2, and S3 should also be optimized for faster computation. As discussed in Section 4.1, many parallelisms that exist in S1, S2, and S3 can be exploited by PL. For example, the main operation of S1 and S2 is convolution, which can be operated in parallel with each pixel being processed per cycle as discussed in Chai and Shi [2011], Fowers et al. [2012] and Mohanty and Meher [2013]. S3, which contains multiple multiplications and additions, can also be easily implemented in hardware. After optimization, stages S1 to S4 can work in a streaming mode with the same delay for each stage.

4.2.3. Optimizing Memory Access. As mentioned in the previous subsection, the majority of the execution time becomes the time to move the output flow tensors to the input memory for the next iteration when separate memory sections for input and output are fixed to the VDMA. To avoid data transference between different memory sections, a structure for memory switching is designed based on ZYNQ. As shown in Figure 7, there are four high-performance ports (i.e., VDMA0-MM2S, VDMA0-S2MM, VDMA1-MM2S, and VDMA1-S2MM) with read and write FIFOs and two dedicated memory ports on the DDR controller based on the AXI protocol on ZYNQ. In our work, two memory sections (i.e., *Mem0* and *Mem1*) are used for memory switching, which are connected with VDMA0 and VDMA1, respectively, during different iterations. For instance, during one iteration, the input data are fetched from *Mem0*. After processing, the output results are written into *Mem1*. During the next iteration, the input data are fetched from *Mem1* (which is updated in the previous iteration). Then, the output results are written into *Mem0*, which will be used as the input data for the next iteration. This scheme avoids moving data between memory sections, and the execution time is reduced accordingly.

5. SYSTEM DEVELOPMENT ON HARDWARE AND SOFTWARE

We implement the system on Digilent ZedBoard, an evaluation platform for the Xilinx ZYNQ architecture, which is introduced in Section 2.3.

The system is based on Linux, and the layered architecture is illustrated in Figure 8. The Xilinx Vivado HLS is taken as the development tool [Vivado 2015]. In general, the system development includes software design and hardware design. The software runs in the PS part of ZYNQ, and the hardware is primarily implemented in the PL part, except for some configurations set on PS. The software design focus

Fig. 7. Optimized data stream for iteration on ZYNQ.

Fig. 8. Layered architecture of the system.

on (1) the applications and (2) device drivers. The hardware design focus on (3) the FPGA-based accelerators. As mentioned in Section 2.3, in addition to the IP core of the main task (optical flow computation), the IP core of AXI-VDMA is also required to handle the image data. The detailed programming guide of VDMA can be found in its manual [VDMA 2015]. The application varies according to different real-life scenarios. In this article, we simply compose a test program to record the performance for further analysis. The device drivers provide the operation handler of the hardware to the applications by accessing the registers in the control path. In this article, we focus on the development of the hardware of FPGA-based accelerators for variational optical flow computation.

Figure 9 presents the detailed design of the FPGA (PL) part. In this system, the data path is encapsulated into the AXI4-Stream, and the control path is encapsulated into AXI4-Lite, which are both based on the AXI4 bus protocol. The AXI4 bus connects the optical flow IP core with the physical devices, such as memory and processor. It creates

Fig. 9. Detailed design of FPGA.

a great portability: the same IP core can be applied to various platforms, provided that there is an AXI4 bus.

We use the C language rather than low-level HDL to realize the optical flow IP core for higher development efficiency. However, the peripheral interfaces (e.g., memory controller, AXI ethernet, AXI PCIe) are still realized with low-level RTL HDL because special requirements in the timing constraints, wiring requirements, architecture, and other aspects of these underlying hardware are essential. Fortunately, the IP library provides most IP cores of these standard devices.

As an example of the implementation of optical flow IP, Figure 10 illustrates the hardware structure of CBG. This is a detailed description of Figures 2 and 3. The input data are *img1_int8* and *img2_int8*, which are the current pixels of two frames of images, respectively, and the initialized optical info (*inu_int32* and *inv_int32*). The outputs are the updated optical flows (*outu_int32* and *outv_int32*). The postfix, such as *_int8*, gives the tip of data type, where *int8* means an 8-bit integer. As mentioned earlier, the input data are packaged as AXI4-Stream slave mode, and the output data are packaged as master mode.

During the first three stages, to save resources and retain the precision, fixed-point data are adopted for calculation. Thus, the float convolution mask is transformed to fixed-point by multiplying by some integer. In the last stage, the iterative calculation is a fine-grained approximation. Therefore a floating-point number is necessary for iterative computation by dividing the number that has been previously multiplied.

6. EXPERIMENTS AND EVALUATIONS

In this section, we evaluate the optimization and acceleration effect and analyze the relationships among various factors. First, we establish the experimental environment. Second, we implement two representative models, HS and CBG by pure software on a general CPU, to obtain the basic data of accuracy and execution time. Third, we implement the optimization methods discussed in Section 4.2 incrementally and evaluated the optimization effect. Fourth, we analyze the relationship and effects of various aspects, such as accuracy, iteration numbers, execution time, and image sizes with different algorithms (HS and CBG). Moreover, we discuss the hardware resources and power consumption. Finally, we compare and evaluate our work with other FPGA-accelerated dense optical flow implementations.

Fig. 10. Hardware structure of CBG.

6.1. Experimental Environment

Unless explicitly specified, the experiments are based on the following hardware platform, image source and algorithm parameters.

6.1.1. Hardware Platform. We select the Digilent ZedBoard with a Xilinx ZYNQ-7020 chip as the uniform platform for the pure software implementation and for our FPGA-accelerated implementation. The PS part is ARM Cortex-A9 (667MHz), which has two 32KB L1 caches, 512KB shareable L2 cache, and 512MB memory. The PL part is an FPGA of Z-7020CLG484 (83.3MHz). For the pure software version, only the PS part is used. For the FPGA-accelerated version, both PS and PL are used.

6.1.2. Image Source. In most real-life situations, the image is captured by a camera and transferred to the PL part for processing directly. However, to fairly compare and evaluate different algorithms and implementations, standardized image sequence are used as the benchmark. In this article, we select the benchmark images from the Middlebury dataset [Baker et al. 2011; Middlebury 2015]. The images are stored as files on an SD card. They were read by PS part and transferred to the PL part.

Fig. 11. Example frames of “Grove2”: (a) frame 10; (b) frame 11.

Table II. Parameter Configurations for HS and CBG

Parameter	Value
Number of initial frames	2
Filter size for presmoothing	7
Filter size for derivative computing	5
Smoothness weight α	100
Data weight γ_1	1
Data weight γ_2	0.01
Relaxation parameter w	1.8
Linear iteration number k	100

In our experiment, frames 10 and 11 from the image sequence *Grove2* (with a resolution of 640×480 pixels) in package “other-gray-two-frames” are used, which are shown in Figure 11. From the figure, we see that the scene is quite complex. As camera moved up, pixels move accordingly and shadows and occlusions occurs.

6.1.3. Algorithm Parameters. The experiment also evaluates the linear HS and CBG algorithms. The configurations of their parameters for optical flow estimation are shown in Table II.

6.2. Performance of HS and CBG as Pure Software

To investigate the performance of optical flow computation, we implemented two representative models, HS and CBG, on the PS part of ZYNQ as pure software (denoted as “SW”) to compare their accuracy and execution time.

6.2.1. Execution Time. Figure 12 shows the execution times of HS and CBG, where “I/O” refers to the time for loading and saving the image files from the SD card. The execution times of both HS and CBG on the embedded processor are clearly too long (over 20 seconds per frame) to be real time. The execution time of CBG is slightly longer (9.6%) than that of HS. Notably, the iteration scheme (S4) spends the largest portion of the execution time irrespective of which algorithm is used.

6.2.2. Accuracy. In this article, *Average Angular Error* (AAE) is used to measure the accuracy of the optical flow [Baker et al. 2011; Barron et al. 1994]. *Angular Error* (AE) is defined in Equation (2), where (u_r, v_r) is the referential ground-truth value. AAE is the AAE of all pixel over the entire frame. The smaller the AAE, the better the accuracy.

Fig. 12. Execution time of variational optical computation on CPUs.

Fig. 13. Accuracy of variational optical flow computation on CPUs.

Figure 13 shows the AAE comparison between HS and CBG. As shown in this figure, the accuracy improves with the iteration number; however, this corresponds to more execution time. When the same iteration number is used, CBG is generally more accurate than HS. This result means that HS requires substantially more iterations than CBG to obtain the same accuracy and diminishes its advantages in terms of execution speed. As mentioned in Section 2, most related work focusing on accelerating dense optical flow computation on FPGA-based platforms selected HS as the representative model. It was easy to implement but is less accurate because fewer iterations were used in those works to demonstrate the fast speed.

$$AE = \arccos \left(\frac{1 + u \cdot u_r + v \cdot v_r}{\sqrt{1 + u^2 + v^2} \sqrt{1 + u_r^2 + v_r^2}} \right) \quad (2)$$

6.3. Effect of Optimization on CBG

Figure 14 demonstrates that by implementing the optimization techniques mentioned in Section 4.2 incrementally, the CBG execution time decreases from 26.68 seconds to 0.82 seconds, corresponding to an acceleration of 32×.

6.4. Analyses of the Performance and Key Factors

In this section, we discuss the performance (execution time and accuracy) with different key factors, such as iteration number, image size, and so forth. For conciseness, we

Fig. 14. Execution time of CBG improved by all the methods. M0: Pure software. M1: accelerate S4 iteration by hardware architecture. M2: loop unrolling, where M2.1 unrolls 2 duplications of the iteration body, and M2.2–3 duplications, M2.3–4 duplications. M3: extending the iteration body. M4: optimize memory access.

Fig. 15. AAE and execution time on different iteration numbers.

denote the most optimized implementation on ZYNQ as “SW-HW” (software-hardware cooperated version).

6.4.1. Iteration Number. In our implementation, we use 32-bit fixed-point computation in pre-smoothing (S1) and derivative computing (S2). The kernel for pre-smoothing is [4, 8, 12, 16, 12, 8, 4], and that for derivative computing is [1, -8, 0, 8, -1]. Single precision floating-point computation is deployed in motion tensors construction (S3) and the iteration body (S4) to retain accuracy.

Figure 15 shows the execution time and accuracy along with the iteration number of the most optimized CBG on ZYNQ. The accuracy difference between SW-HW and SW is caused by the red-black scheme and the fixed-point arithmetic used in SW-HW. In general, with the same number of iterations, SW has more accurate results than SW-HW. However, the SW-HW CBG is able to improve accuracy with more iterations. For example, the SW-HW CBG with 400 iterations outperforms the SW CBG with 100 iterations in terms of both speed (1.92s vs. 26.68s) and accuracy (AAE 12.82 vs. 14.61). Our experimental results show that for a fixed AAE of 14.61, SW-HW CBG requires 310 iterations and the execution time is 1.57 seconds.

6.4.2. Image Sizes and Duplication Number of S4. Figure 16 shows the execution times of SW-HW CBG for different image sizes, with different duplications of S4. Increasing the S4 duplication number clearly reduces the execution time. Furthermore, two trends can be observed from Figures 16(b) and 16(c): (1) For a given size, the execution time per pixel decreases with more S4 duplications. However, the speed-up decreases with

(a)

(b)

(c)

Fig. 16. Relationship between execution time and image size. (a) Execution time and image size; (b) execution time per pixel and image size (fixed image size, compare with different duplications); (c) execution time per pixel and image size (fixed duplications, compare with different image sizes).

more duplications being employed. For instance, the execution times per pixel of the 640×480 image are $6.4\mu s$, $4.0\mu s$, $3.0\mu s$, and $2.7\mu s$ when the duplication number of S4 increases from 1 to 4. However, the speed-up as the duplication number increases from 2 to 4 is 1.6, 1.33, and 1.15, respectively. (2) For a given specific duplication number of S4, the execution time per pixel decreases in general when the image size increases due to better pipeline efficiency. For instance, the processing time per pixel with one duplication of S4 decreases from $14.4\mu s$ to $6.4\mu s$ when the image resolution increases from 316×215 to 640×480 and from $4.6\mu s$ to $2.7\mu s$ with four duplications of S4.

Table III. Time Performance Comparison between SW and SW-HW

Image	Size	SW [s]	SW-HW [s]	speed-up
Yos.	316×252	5.16	0.37	13.95
Ven.	420×380	10.99	0.48	22.90
Rub.	584×388	16.83	0.71	23.70
Gro.	640×480	26.68	0.82	32.54

Fig. 17. Execution time per pixel of SW and SW-HW.

Fig. 18. Time performance comparison between HS and CBG on ZYNQ.

Table III compares the execution times between SW and SW-HW. Then, Figure 17 illustrates the execution time per pixel of SW and that of SW-HW. For SW, the execution time per pixel increases as the image size increases because more intermediate data cause more cache misses. In contrast, as mentioned previously, for SW-HW, the execution time per pixel generally decreases when the image size increases because of better pipeline efficiency. As shown in Figure 17, as the image size increases, the execution time per pixel of SW increases from $64.7\mu\text{s}$ to $86.8\mu\text{s}$ while that of SW-HW decreases from $4.6\mu\text{s}$ to $2.7\mu\text{s}$. This result indicates that the hardware-accelerated technique should be used for real-time processing, particularly for larger images.

6.4.3. HS vs. CBG. Figure 18 shows the execution times of FPGA-accelerated HS and CBG on ZYNQ with different duplications of S4. The image size being tested is 640×480 . From this figure, we can observe that (1) the execution time decreases

(a)

(b)

Fig. 19. Resource utilization. (a) Resource estimation of optical flow IP core designed by HLS; (b) slice logic utilization on ZYNQ.

(e.g., from 1.76s to 0.61s) with more duplications of S4 implemented on PL, and (2) the additional time to compute the second-order derivative is masked by the pipelining in SW-HW. This result indicates that the overhead that CBG has over HS reduces from SW to SW-HW, as shown in Figure 18. Thus, CBG is more preferable for hardware-accelerated schemes.

6.5. Resource Utilization and Scalability

Figure 19(a) shows the synthesis report on the resource utilization of the optical flow IP core designed by HLS, while Figure 19(b) shows the report of implementation of the entire system including VDMA on ZYNQ. As shown in the figure, all types of resource utilization increase as the duplication number of S4 increases. For 4 duplications of S4,

Fig. 20. Execution time and resource utilization of K7. (a) Execution time of ZYNQ and K7; (b) resource utilization of K7.

Fig. 21. Power consumption of ZYNQ SoC.

over 90% LUT and BRAM are utilized, which limits the parallelism of the algorithm from being exploited further.

From the current hardware platform (ZedBoard with ZYNQ-7020), with the most optimized acceleration, it costs 0.61 seconds to process a 640×480 image. (The 0.21 second I/O time is not included.) It may still be slightly too slow for some real-time applications. As shown in Figure 16, using more duplications of S4 (loop unrolling) will reduce the execution time. However, it consumes more resources.

To improve the performance, we perform an additional experiment on a new hardware platform. A Kintex-7 (XC7K325T-2FFG900C, 150MHz)-based PC-FPGA platform is used to provide more FPGA resources. Because the implementation of optical flow computation is AXI-compliant and an AXI-PCIe bridge is implemented by us, the implementation can be ported to the new platform without major modifications. We conduct 8 duplications of S4 on Kintex-7 (K7) to improve the parallelism. As shown in Figure 20, the execution time of optical flow computation excluding the I/O time is decreased to 0.085 second when 81% of the LUT resources are consumed in Kintex-7 FPGAs.

6.6. Power Consumption

Figure 21 shows the power consumption of ZYNQ with the SW-HW optical flow system implementation, as evaluated with the Vivado tool [Vivado 2015]. Obviously, the

Table IV. Comparison with Other FPGA-Accelerated Implementations

Work	Hardware	Model	Lang.	F(MHz)	Size	Iter	FPS	PPS(K)	AAE	Power(W)	Energy(J)
i7-HS	CPU: Intel i7	HS	C	2900	640x480	100	0.43	134	18.56	35	80.5
i7-CBG	CPU: Intel i8	CBG	C	2900	640x480	100	0.38	116	14.61	35	92.3
A9-CBG	CPU: ARM: Cortex A9	CBG	C	667	640x480	100	0.04	13.3	14.61	1.308	30.1
ZYNQ-HS	FPGA-SoC: Zynq 7020	HS	C	83.3	640x480	100	1.72	530	18.56	1.848	1.07
ZYNQ-CBG	FPGA-SoC: Zynq 7020	CBG	C	83.3	640x480	100	1.64	503	23.26	1.881	1.15
ZYNQ-CBG2	FPGA-SoC: Zynq 7020	CBG	C	83.3	640x480	310	0.74	226	14.61	1.881	2.55
K7-CBG	FPGA-SoC: Zynq 7020	CBG	C	150	640x480	100	11.8	3614	23.26	4.436	0.38
K7-CBG2	FPGA-SoC: Zynq 7020	CBG	C	150	640x480	310	5.88	1807	14.61	4.436	0.75
Arribas	FPGA	HS	HDL	N/A	50X50	3	19	4.75	N/A	N/A	N/A
Martin	FPGA	HS	HDL	N/A	256x256	N/A	60	3840	N/A	N/A	N/A
Rustam	FPGA	HS	HDL	105	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Gultekin	FPGA	HS	HDL	50	256x256	1	257	16842	1.319	0.844	N/A
Bahar	FPGA: Altera Cyclone II	HS	HDL	12.5	240*320	8	1030*	734.3	N/A	N/A	N/A
Komo	FPGA: Xilinx Virtex 7	HS	HDL	148.5	1920*1080	128	84	175000	16.51	12.22	N/A
Kunz	FPGA: Altera Stratix IV	HS	HDL	294.9	1920*1080	30	60	124416	N/A	N/A	N/A

*: The high FPS of Bahar is a result of the sparse computing, which is different with other works.

References: Arribas: Arribas and Macia [2003]; Martin: Martín et al. [2005]; Rustam: Rustam et al. [2012]; Gultekin: Gultekin and Saranlı [2013]; Bahar: Bahar and Karimian [2012]; Komo: Komorkiewicz et al. [2014]; Kunz: Kunz et al. [2014].

667MHz PS consumes 1.308W, more than 70% power of the entire system. The static power consumption of the device and the power consumption of PS remain the same across all implementations; only the power consumption of PL varies.

As shown in Figure 21, two factors affect the power consumption of PL. One is resource utilization. When the duplication number of S4 changes from 1 to 4, the power consumption of PL at an 83.3MHz working frequency changes from 0.356W to 0.414W. The other factor is the working frequency of the system. For *Dup1*, the power consumption changes from 0.356W to 0.425W when the working frequency changes from 83.3MHz to 100MHz. For *Dup2*, the power consumption changes from 0.37W to 0.436W. Regardless, our experiments show that ZYNQ SoC is able to provide power-efficient computation for embedded applications. For the K7-Dup8 implementation, the power consumption of the PL and I/O is 4.436W, which is considerably higher than that of the ZYNQ-Dup4 implementation. However, because the processing time is drastically decreased, the energy consumption is low.

6.7. Comparison with Other Implementations

To compare the performance on the heterogeneous SoC with that on a mainstream computer, we implement the pure software versions of HS and CBG on a PC with an i7-3520M CPU (2.9 GHz). For HS (denoted as i7-HS), the execution time is 2.3 seconds per frame of a 640×480 image, and the power consumption is 35W, which means 80.5J of energy. For CBG (denoted as i7-CBG), the execution time is 2.635 seconds with the same power; therefore, 92.3J of energy is consumed.

Table IV shows the comparison among the implementation in this article. The important parameters are presented clearly. In addition to frames per second (fps), we also use pixels per second (PPS) to indicate the processing speed because the image

resolution varies among all implementations. Note that the time to load the image and to save the flow result is not included for computing PPS. The “Power” and “Energy” columns indicate the power and energy consumed for a frame, not a pixel.

In this table, we list 2 cases for the implementation of CBG both on Zynq and K7: (i) for the same iteration number (100) as others and (ii) for the same AAE as the pure software CBG (14.61). From this table, we can observe that with the same accuracy, the CBG on ZYNQ (ZYNQ_CBG2) is faster than that on the i7 (i7_CBG), and the energy consumption of the former is quite lower than the latter. Other comparisons can be derived from this table.

To provide more information, we also included other FPGA-accelerated dense optical flow implementations in this table. Note that it is difficult or even impossible to compare the different implementations fairly because they are quite different in terms of hardware architecture, working frequency, resource used, image size and content, density, iteration numbers, accuracy, power consumption, and so forth. For example, the table shows that Gultekin is faster in PPS than our work even with lower frequency hardware. However, this speed difference is primarily a result of the different number of iterations. Gultekin only iterates once, whereas our implementation iterates hundreds of times for a higher accuracy. However, the AAE of Gultekin is very small. This point deserves more explanation and analysis. AAE is the average angular error of all pixels over the entire frame. The camera for Gultiken’s test image sequence was fixed, and only a small number of pixels (the outline of the moving objects) moved between two consecutive frames. Therefore, according to Equation (2), $AE = 0$ because $u = 0$, $v = 0$, $u_r = 0$, and $v_r = 0$ on these fixed pixels. Because the number of fixed pixels is considerably greater than the number of moved pixels, the AAE becomes very small. In the contrast, our camera was moved between the two consecutive frames. As a consequence, almost all pixels moved, and even shadows and occlusions changed accordingly. Therefore, our AAE is higher than that of Gultiken. Therefore, AAE should be compared based on the same test image. However, as mentioned in Section 6.2, CBG usually performs better than HS for the situations of natural scenes that slight changes in brightness occurs, particularly for scenarios where the camera is moved by different frames.

As shown in the table, the basic HS model was generally selected in other FPGA implementations due to its simplicity. Most of these works that we use for comparison use low-level HDL to describe their algorithms, which is a completely hardware design work that requires long development time.

7. CONCLUSION

Optical flow computation plays an important role in real-world computer vision systems. Researchers have developed various algorithms to deal with different situations. These algorithms can be divided into three models: HS, CLG, and CBG. For those applications requiring high performance and low power consumption, FPGA is generally considered a suitable computing platform. However, although CBG is more suitable for some natural scenes, due to the complexity of HDL programming, only the simplest HS model are widely implemented by various FPGA platforms.

This article provided a common workflow of the variational optical flow computation that is suitable for all the algorithms belonging to the three categories of models. As a representative case, this article proposed an implementation of the CBG model optimized on the Xilinx ZYNQ FPGA SoC and demonstrated the use of C rather than HDLs to implement a high-quality optical flow algorithm on a FPGA-accelerated heterogeneous SoC. Our implementation obtained two major improvements. First, the code development process for FPGA is significantly simplified. The HLS tools relieve engineers of nontrivial low-level details. It is more efficient and produces more robust

codes. Second, the computation speed of the optical flow is significantly improved. We achieved a 32-fold improvement in the computation speed compared with the pure software approach while retaining a low energy consumption. Notably, the implementation also runs faster than a mainstream PC while consuming significantly less energy.

The results demonstrate that off-the-shelf commercial FPGA SoCs coupled with development tools have the potential to provide a high-performance and high-efficiency platform for real-world embedded computer vision systems. The proposed optimization provides a very useful solution for implementing the middle-level optical flow algorithms on FPGA-based platforms.

REFERENCES

- Altera. 2015. Altera SoCs: When architecture matters. Retrieved April 21, 2016 from <http://www.altera.com/devices/processor/soc-fpga/overview/proc-soc-fpga.html>.
- Tomer Amiaz and Nahum Kiryati. 2006. Piecewise-smooth dense optical flow via level sets. *Int. J. Comput. Vis.* 68, 2 (2006), 111–124.
- Pedro Cobos Arribas and Felix Monasterio Huelin Macia. 2003. FPGA implementation of Santos-Victor optical flow algorithm for real-time image processing: An useful attempt. In *Microtechnologies for the New Millennium 2003*. International Society for Optics and Photonics, 23–32.
- Mohammad Reza Balazadeh Bahar and Ghader Karimian. 2012. High performance implementation of the Horn and Schunck optical flow algorithm on FPGA. In *Proceedings of the 2012 20th Iranian Conference on Electrical Engineering (ICEE'12)*. IEEE, 736–741.
- Simon Baker, Daniel Scharstein, J. P. Lewis, Stefan Roth, Michael J. Black, and Richard Szeliski. 2011. A database and evaluation methodology for optical flow. *Int. J. Comput. Vis.* 92, 1 (2011), 1–31.
- John L. Barron, David J. Fleet, and Steven S. Beauchemin. 1994. Performance of optical flow techniques. *Int. J. Comput. Vis.* 12, 1 (1994), 43–77.
- Berkeley Design Technology, Inc. 1994. An independent evaluation of: The AutoESL AutoPilot high-level synthesis tool. Retrieved April 21, 2016 from <http://www.bdti.com/MyBDTI/pubs/AutoPilot.pdf>.
- Michael J. Black and Paul Anandan. 1996. The robust estimation of multiple motions: Parametric and piecewise-smooth flow fields. *Comput. Vis. Image Underst.* 63, 1 (1996), 75–104.
- Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. 2010. State-of-the-art in heterogeneous computing. *Sci. Program.* 18, 1 (2010), 1–33.
- Thomas Brox, Christoph Bregler, and Jagannath Malik. 2009. Large displacement optical flow. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'09)*. IEEE, 41–48.
- Thomas Brox, Andrés Bruhn, Nils Papenberg, and Joachim Weickert. 2004. High accuracy optical flow estimation based on a theory for warping. In *Computer Vision-ECCV 2004*. Springer, 25–36.
- Thomas Brox and Jitendra Malik. 2011. Large displacement optical flow: Descriptor matching in variational motion estimation. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 3 (2011), 500–513.
- Andres Bruhn and Joachim Weickert. 2005. Towards ultimate motion estimation: Combining highest accuracy with real-time performance. In *Proceedings of the 10th IEEE International Conference on Computer Vision (ICCV'05)*, Vol. 1. IEEE, 749–755.
- Andrés Bruhn, Joachim Weickert, Christian Feddern, Timo Kohlberger, and Christoph Schnorr. 2005b. Variational optical flow computation in real time. *IEEE Trans. Image Process.* 14, 5 (2005), 608–615.
- Andrés Bruhn, Joachim Weickert, and Christoph Schnorr. 2005a. Lucas/Kanade meets Horn/Schunck: Combining local and global optic flow methods. *Int. J. Comput. Vis.* 61, 3 (2005), 211–231.
- Zhilei Chai and Jianbo Shi. 2011. Improving KLT in embedded systems by processing oversampling video sequence in real-time. In *Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig'11)*. IEEE, 297–302.
- Haiyang Chao, Yu Gu, and Marcello Napolitano. 2014. A survey of optical flow techniques for robotics navigation applications. *J. Intell. Robot. Syst.* 73, 1–4 (2014), 361–372.
- Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. 2010. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 225–236.
- D. J. Evans. 1984. Parallel SOR iterative methods. *Parallel Comput.* 1, 1 (1984), 3–18.
- Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. 2012. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 47–56.

- Gokhan Koray Gultekin and Afsar Saranli. 2013. An FPGA based high performance optical flow hardware design for computer vision applications. *Microprocess. Microsyst.* 37, 3 (2013), 270–286.
- Pascal Gwosdek, Andrés Bruhn, and Joachim Weickert. 2010. Variational optic flow on the sony PlayStation 3. *J. Real-Time Image Process.* 5, 3 (2010), 163–177.
- Bruno Herissé, Tarek Hamel, Robert Mahony, and François-Xavier Russotto. 2012. Landing a VTOL unmanned aerial vehicle on a moving platform using optical flow. *IEEE Trans. Robot.* 28, 1 (2012), 77–89.
- Dominik Honegger, Pierre Greisen, Lorenz Meier, Petri Tanskanen, and Marc Pollefeys. 2012. Real-time velocity estimation based on optical flow and disparity matching. In *Proceedings of the 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'12)*. IEEE, 5177–5182.
- Berthold K. Horn and Brian G. Schunck. 1981. Determining optical flow. In *Proceedings of the 1981 Technical Symposium East*. International Society for Optics and Photonics, 319–331.
- Mateusz Komorkiewicz, Tomasz Kryjak, and Marek Gorgon. 2014. Efficient hardware implementation of the Horn-Schunck algorithm for high-resolution real-time dense optical flow sensor. *Sensors* 14, 2 (2014), 2860–2891.
- Michael Kunz, Alexander Ostrowski, and Peter Zipf. 2014. An FPGA-optimized architecture of Horn and Schunck optical flow algorithm for real-time applications. In *Proceedings of the 2014 24th International Conference on Field Programmable Logic and Applications (FPL'14)*. IEEE, 1–4.
- Vincenzo Lippiello and Bruno Siciliano. 2012. Wall inspection control of a VTOL unmanned aerial vehicle based on a stereo optical flow. In *Proceedings 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'12)*. IEEE, 4296–4302.
- Bruce D. Lucas, Takeo Kanade, and others. 1981. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, Vol. 81. 674–679.
- José L. Martín, Aitzol Zuloaga, Carlos Cuadrado, Jesús Lázaro, and Unai Bidarte. 2005. Hardware implementation of optical flow constraint equation using FPGAs. *Comput. Vis. Image Underst.* 98, 3 (2005), 462–490.
- Middlebury. 2015. Datasets of Middlebury. Retrieved April 21, 2016 from <http://vision.middlebury.edu/flow/data/>.
- Yoshiki Mizukami and Katsumi Tadamura. 2007. Optical flow computation on compute unified device architecture. In *Proceedings of the 14th International Conference on Image Analysis and Processing (ICIAP'07)*. IEEE, 179–184.
- Basant Kumar Mohanty and Pramod Kumar Meher. 2013. Memory-efficient high-speed convolution-based generic structure for multilevel 2-D DWT. *IEEE Trans. Circuits Syst. Video Technol.* 23, 2 (2013), 353–363.
- Ruzali Rustam, Nor Hisham Hamid, and Fawnizu Azmadi Hussin. 2012. FPGA-based hardware implementation of optical flow constraint equation of Horn and Schunck. In *Proceedings of the 2012 4th International Conference on Intelligent and Advanced Systems (ICIAS'12)*, Vol. 2. IEEE, 790–794.
- Narayanan Sundaram, Thomas Brox, and Kurt Keutzer. 2010. Dense point trajectories by GPU-accelerated large displacement optical flow. In *Computer Vision—ECCV 2010*. Springer, 438–451.
- Patrik Sundberg, Thomas Brox, Michael Maire, Pablo Arbeláez, and Jitendra Malik. 2011. Occlusion boundary detection and figure/ground assignment from optical flow. In *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'11)*. IEEE, 2233–2240.
- VDMA. 2015. LogiCORE IP AXI video direct memory access v5.02.a. Retrieved April 21, 2016 from http://www.xilinx.com/support/documentation/ip_documentation/axi_vdma/v5_02_a/pg020_axi_vdma.pdf.
- Vivado. 2015. Vivado Design Suite - HLx editions. Retrieved April 21, 2016 from <http://www.xilinx.com/products/design-tools/vivado/>.
- Li Xu, Jiaya Jia, and Yasuyuki Matsushita. 2012. Motion detail preserving optical flow estimation. *IEEE Trans. Pattern Anal. Mach. Intell.* 34, 9 (2012), 1744–1757.
- Henning Zimmer, Andrés Bruhn, and Joachim Weickert. 2011. Optic flow in harmony. *Int. J. Comput. Vis.* 93, 3 (2011), 368–388.
- ZYNQ. 2015. Expanding the all programmable SoC portfolio. Retrieved April 21, 2016 from <http://www.xilinx.com/products/silicon-devices/soc/ZYNQ/index.htm>.

Received October 2015; revised April 2016; accepted May 2016